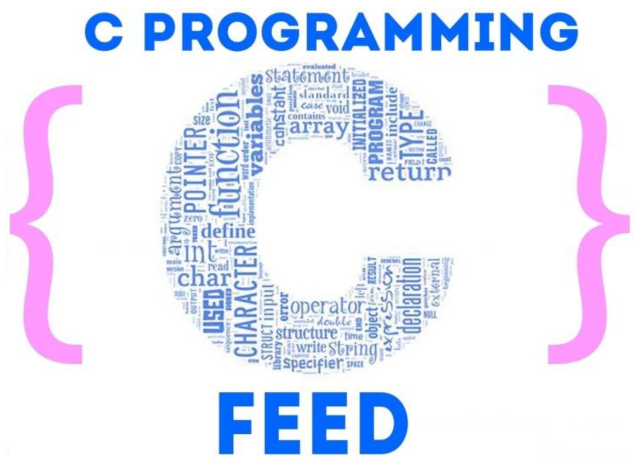
First Step Towards Programming



2015

Author

Arun Reddy Pothireddy

Assistant Professor
Department of Computer Science & Engineering
&
Alumni Chief Coordinator
Christu Jyothi Institute of Technology & Science

Algorithm:

An algorithm is an effective method expressed as a finitelistof well-defined instructions for calculating a function.(or)In Simple words, an Algorithm is defined as the finite number of steps required to perform a task.

Expressing algorithms

Algorithms can be expressed in many kinds of notation, including natural languages, pseudo code, flowcharts, drakon-charts, programming languages or control tables (processed by interpreters).

Pseudo Code: Algorithm written in English language is called Pseudo Code.

Flow Charts:

The Diagrammatic representation of an algorithm is called a flow Chart. There are many polygons to represent a flow chart. Some of the polygons used are mentioned below:

Let me explain you the importance of an algorithm with the help of an example.

Ex: Write the algorithm to find the value of x in the equation ax+b=0

Step1: start (Algorithm starts with the key word "start")

Step 2: Input a,b values (give the values for the a and b)

Step3: Calculate the –b/a

Step 4: store the value in x

Step 5: print the value of x

Step 6: Stop

This is the general way we write a algorithm, but please do look after the output under these cases

Case 1: when a=3 and b=2 the value of x is -3/2

Case2: when a=0 and b=2 the value of x is undetermined……. that means my algorithm is not accepting the value of "0".

By Observing the above two cases we can say that the algorithm is not so effective because it is not accepting all values.

Criteria for the Algorithm:

Input: The Algorithm should accept the each and every input given by the user

Output: It should definitely produce the output .

Effectiveness: Crystal clear approach towards the algorithm gives us the effectiveness.

Finiteness: Every Algorithm should terminate after some finite number of Instructions.

Definiteness: Unnecessary use of variables or effective use of resources gives us the definiteness.

Now, let me Re-Modify the algorithm as:

```
Step1:start

Step2: Input a,b

Step3: if a is not equal to zero and b is equal to zero  then return the value          zero

Step4:if a is not equal to zero and b is not zero then calculate –b/a

Step 5: store the value calculated in x

Step 6 print the value of x

Step5:if a is equal to 0 then print a statement “please enter a non-zero value”

Step6:Stop.
```

Now, the Algorithm is following all the criteria required for the “best algorithm.”

Which level is C language belonging to?

S.no	High Level	Middle Level	Low Level
1	High level languages provides almost everything that the programmer might need to do as already built into the language	Middle level languages don’t provide all the built-in functions found in high level languages, but provides all building blocks that we need to produce the result we	Low level languages provides nothing other than access to the machines basic instruction set

		want	
2	Examples: Java, Python	C, C++	Assembler

C language is a structured language.

S.no	Structure oriented	Object oriented	Non structure
1	In this type of language, large programs are divided into small programs called functions	In this type of language, programs are divided into objects	There is no specific structure for programming this language
2	Prime focus is on functions and procedures that operate on data	Prime focus is on the data that is being operated and not on the functions or procedures	N/A
3	Data moves freely around the systems from one function to another	Data is hidden and cannot be accessed by external functions	N/A
4	Program structure follows "Top Down Approach"	Program structure follows "Bottom UP Approach"	N/A
5	Examples: C, Pascal, ALGOL and Modula-2	C++, JAVA and C# (C sharp)	BASIC, COBOL, FORTRAN

FIRST STEP TOWARDS PROGRAMMING

Let us see the basic C program that involves some simple statements

A C program basically consists of the following parts:

1. Preprocessor Commands

2. Functions

3. Variables

4. Statements & Expressions

5. Comments

For Instance, there is a code written to print a statement “C is so easy to learn”.

```
#include<stdio.h>

int main()

{

  /*My first C program*/

 printf(“C is so easy to
learn”);

return 0;
```

Let us see the description of the program

1. #include<stdio.h>: This is a preprocessor command that includes standard input output header file (stdio.h) from the C library “before” compiling a C program

2. int main(): This is the main function where the program execution begins.

3.{: This Symbol indicates the beginning of the main function

4. /*……*/ are known as comment lines .whatever written in between those comment lines, will be ignored by the compiler.

5. printf command prints the output onto the screen i.e., C is so easy to learn will be appeared on the screen.

6.return 0: This statement terminates the program and returns 0.

Basic Structure of a C program:

1. Documentation section
2. Link Section
3. Definition Section
4. Global declaration section
5. Function prototype declaration section
6. Main function
7. User defined function definition section

S.No	Section	Description
1	Documentation Section	Comments about the program can be given here.They can be written any where in the program Ex:/*first C program*/
2	Link Section or HeaderFile Section	Header files are required to execute the C program. Ex:#include<stdio.h>
3	Definition Section	Variables are defined and values are set to these variables.

4	Global Declaration Section	Global variables are defined in this section.When a variable is to be used through out the program ,can be defined in this section
5	Function prototype declaration section.	Function prototype gives many information about a function like return type, parameter names used inside the function
6	Main Function	Every Program starts its execution from main function and this main function contains two major sections called declaration section and executable section.
7	User defined function Section	User can define their own functions in this section which perform particular task as per the user requirement.

FUNCTIONS

How to define Function

A Function is a self contained block of perpendicular more statements or a sub-program which is design for a particular task is called function. In simple words, Function is a group of statements that together perform a specific task.

Note: Every C program has at least one Function i.e. main( )

You can divide up your code into separate functions. How you divide up your code among different functions is up to you but logically the division should be in such a way that each function performs a specific task. The C function contain set of instructions enclosed by { } braces.

Importance of Function

- The basic purpose of the function is code reusability.
- By using functions, we can develop an application in module format. Thus, it provides modularity to the program.
- When we are developing the program in module format then easily we can debug the program.
- By using functions we can reduce the coding part.
- By using functions we can keep track of what they are doing.
- C-program is a collection of functions and from any function we can invoke (call) any other function.

Syntax for function :

```
return_type   function_name(parameter list) /*function header*/
        {
             Statement Block ;       /* function body */
              return statement;
         }
```

A *Function declaration* tells the compiler about a function's name, return type and parameter.

A *Function definition* provides the actual body of the function.

- Return type
  The return_type is the data type or the keyword of the value the function returns.
  - In implementation whenever a function doesn't returns any value back to the calling place then specify the return type as '*void'* i.e no return value
  - In implementation whenever a function returns other than void then specify the return type as return value type i.e. one type of return value it is returning same type of return statement should be mentioned.
  - Default return type of any function is an ' int '.
- Function name
  This is the actual or real name of the function. The function name and the parameter list together constitute the function signature.
-
- Parameters
  A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function.
  - Parameters are optional i.e. a function may contain no parameters.
- Function body
  The function body contains a collection of statements that define what the function does.

- → Return type parameters and return statements are optional.
- → In implementation whenever a function returns the value then specifying the return statement is optional. In this case compiler will give a warning message called '*call function should return a value'*.
- → When the function is returning the value, then collecting the value is also optional.

Note: Functions in C language cannot be defined inside the body of another Function.

Array -Definition:

C-Array can be defined as the collection of variables belonging to the same data type.

The array can also be defines in the following manner for better understanding,

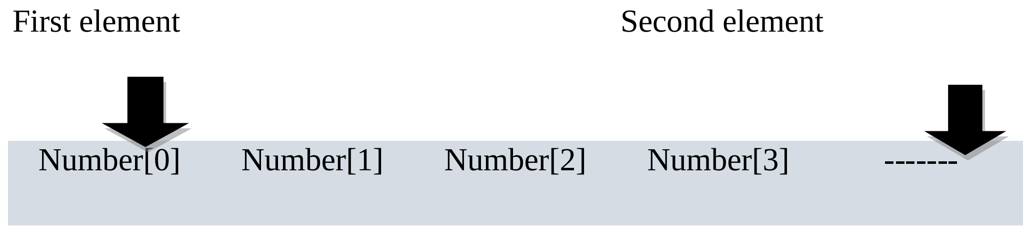
An array is defined as finite ordered collection of homogenous data, stored in contiguous memory locations.

Here the words,

- Finite *means* data range must be defined.
- Ordered *means* data must be stored in contiguous memory addresses.
- Homogenous *means* data must be of similar data type.

What are Arrays?

- ✓ Arrays can also be defined as collection of similar variables. However, arrays in C language are referred as structured data types.
- ✓ When we are working with arrays always memory will be created in contiguous location, so randomly we can access the data.
- ✓ The lowest address corresponds to the first element and the highest address to the last element.



Declaration of Arrays:

```
Datatype arrayName[size];
```

Now let us see the few points about how arrays can be declared:

- ➔ Size of the array must be declared initially. If size of the array is not mentioned, there will be an error from compiler.
- ➔ Always size of the array must be an unsigned integer value which is greater than ‘0’

Example:

int arr[5];

Here size=5 and Sizeof(arr) will be 10Bytes i.e. (5*2B=10B).However, int specifies the type of the variable.

Note:

i. In declaration of the array, size must be unsigned type whose value is greater than 0.
ii. All the elements present in an array must be of same data type.

STRING

iii. Definition of string:
iv. The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.
v. The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello”.
vi. char greeting[6]={‘H’, ’e’, ’l’, ’l’, ’o’ , ’\0’};
vii. If you follow the rule of array initialization then you can write the above statement as following:
viii. char greeting[ ]= ”Hello”;
ix. Syntax:

```
char  str[SIZE];
```

x. *Following is the memory presentation of above-defined string in C/C++:*

H	*e*	*L*	*L*	*o*	*'\0'*

xi.

xii. Actually, you do not place the null character at the end of a string constant. The C complier automatically places the '\0' at the end of a string when it initializes the array. Let us try to print above mentioned string:

xiii. *Program :*

```
#include<stdio.h>

int main()

{

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

printf("Greeting message: %s\n", greeting);

return 0;

}



Output:

  Greeting message:"Hello"
```

C arrays allow the user to define type of variables that can hold several data items of the same kind but structure is another user defined data type available in C programming, which allows the user to combine data items of different kinds.

WHY TO PREFER STRUCTURES?

In C programming, data types are classified into 3 types. They are

1) Primitive data types – int,float,char,double
2) Derived data types - pointer, array, string
3) User defined data types – structure, union, enum
   - All predefined and derived data types are designed to work with basic data elements like integer or character type only.
   - But in real world each information will be there in object format i.e., it contains their own properties and behaviours.

- None of predefined and derived data types support real time object information.
- In implementation whenever a predefined or derived data types are not supporting user requirement then go for user defined data types.
- By using structures we can create our own data types i.e., user defined data types.

DEFINITION:

Structure is a user defined data type which is used to store heterogeneous data items under a unique or common name. Keyword ‘struct’ is used to declare structure. Variables which are declared inside the structure are called Members of Structure.

- Structure is a combination of primitive and derived data type members.C programming language structure contains data members only whereas in C++ data members and member functions also.
- The size of structure is the sum of all member variable size. The least size of structure is 1 byte. In C programming, empty structures are not possible.

How to define a Structure:

To define a structure, one must use the struct statement. The struct statement defines a new data type, with more than one member for user program.

SYNTAX:

```
struct tagname

{

  Data type   member1;

  Data type   member2;

  ………………………………

  ………………………………

};
```

At the end of structure creation (;) must be placed because it indicates an entity is constructed. You can specify one or more structure variables before semicolon but it is optional.

UNIONS

Definition:

A union is a collection of similar or different data types. Unions are the derived data types similar to the structures, but are involved in different activities.

Syntax:

```
union tagname
        {
           data_type mem1;
            data_type mem2;
           ….
            …
         };
```

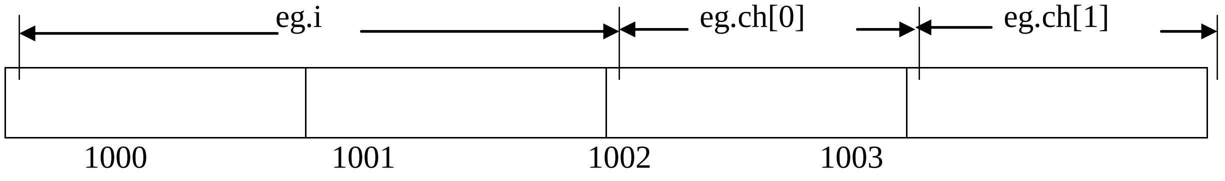
We can access the union elements using a '.' (the dot operator), just the same way in which the structure elements are accessed.

Difference between union and structure:

The major difference between structure and union is 'storage'. In structure, each member has its own location. Whereas in union, all the members share the same location. Let's compare the two data types with an example each.Consider the following example.

```
struct a

{

    short int i;

    char ch[2];

};

struct a eg;
```

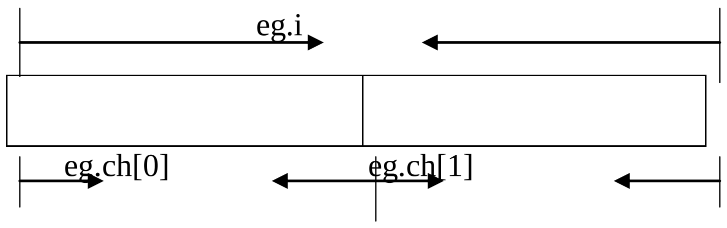
In the example, we defined a structure. This data structure would occupy 4 bytes of memory: 2 bytes for the integer (eg.i) and 1 byte each for the characters (eg.ch[0] and eg.ch[1]).



Now let’s declare a similar data type but instead of using struct, let’s use union. Consider the following example.

```
union a

{

      short int i;

      char ch[2];

};

struct a eg;
```

Now, the memory representation of the above data type is shown in the figure below.



In the example, the union occupies only 2 bytes of memory while the structure occupies 4 bytes. Note that the same memory locations, which are used for eg.i, are also being used by eg.ch[0] and eg.ch[1]. It means that the memory location used by eg.i can also be accessed by eg.ch[0] and eg.ch[1] i.e. we can access two bytes simultaneously.

Points to remember:

- You can define a union with many members, but only one member can contain a value at any given time. Union can handle only one member at a time.
- While structure enables us treat a number of different variables stored at different places in memory, a union offers a way for a section of memory to be treated as a variable of one type on one occasion, and as different variable of different type on another occasion.
- Unions provide an efficient way of using the same memory location for multi-purpose. The memory occupied by union will be large enough to hold its largest member.
- The disadvantage of union is that altering the value of any member affects the other members.

Searching and Sorting

Searching

Common Searching Algorithms

- Linear Search
- Binary Search
- Hash Coding
- Lots more

- Searching requires a key field (e.g., name, ID, code) which is related to the target item.
- When the key field of a target item is found, a *pointer* to the target item is returned. The pointer may be an address, an index into a vector or array, or some other indication of where to find the target.
- If a matching key field isn't found, the user is informed.
- Important factor: Speed! How long does it take to find the target? What is the response time?
- Response time may depend on:
  - size of the list (number of records and record size)
  - data structure used (vector, linked list, binary tree, etc.)
  - data organization (key ordered, random, etc.)
  - search strategy (linear, binary, other)
  - location of the list
    - external (on a disk or other device)
    - internal (in memory)
  - and more
- Search time can be measured with *big-O* notation:

O(n) linear time

O(log2n) logarithmic time

O(1) constant time

What does this mean? Assume you have n items in list.

linear time: It takes about n probes to find target. Double the size of the list and you double the number of probes.

logarithmic time: It takes about log2(n) probes. Double the list size and increase the number of probes by one. Double it again and increase the probes by only one.

constant time: The search time is independent of the size of the list. Double the list size and the number of probes remains the same.

What if we cut list in half?

- Linear Search - Array based list

  Scan the list until:

  1. last record is searched, or
  2. target record is found

  Return:

  3. index of found target or other useful information (e.g., find a name and return the phone number), or
  4. flag (e.g., -1) indicating target record not found

  Example

```
//---------------------------linear search----------------------
//
//   Search A[N] for the first instance of a target.
//
//   Input:  an initialized array of N integers
//   Return value:  array index of the first instance of target,
//                  or -1 if target is not in the array
//--------------------------------------------------------------

int lsearch(int A[], int target)  {

  int index = 0;

  while(index < N && A[index] != target)
    index++;

  if(index < N)  return index;
  else  return -1;
}
-OR-

int lsearch(int A[], int target)  {

  int index;

  for(index = 0; index < N && A[index] != target; index++);

  if(index < N)  return index;
  else  return -1;
}
```

If A holds the following values, what would be returned if target is 0?

```
     +---+---+---+---+---+---+---+---+---+
  A: |13 | 4 | 6 | -8| 0 | 2 | 0 | 9 | 6 |
     +---+---+---+---+---+---+---+---+---+
       0   1   2   3   4   5   6   7   8
```

How would you change the code to return the numbered position instead of the array index? That is, return 5 instead of 4 in the example above.

Example   Search for the last occurrence of target.

```
int lsearch3(int A[], int target)  {
  int index = N-1;

  while(index >= 0 && A[index] != target)
    index--;

  if(index >= 0)  return index;
  else  return -1;
}
```

-OR-

```
int lsearch4(int A[], int target)  {
  int index;

  for(index = N-1; index >= 0 && A[index] != target; index--);

  if(index >= 0)  return index;
  else  return -1;
}
```

Note:

- We usually search for a key within a large record.
- Linear search is easy to implement, but is slow.

How many probes are needed on a randomly ordered list of size n?

worst case___________________

best case____________________

average case_________________

Binary Search   O(log2(n))

Binary search is fast, but the list must be ordered.

Algorithm

0. Probe middle of list
   1. If target equals list[mid], FOUND.
   2. If target < list[mid], discard 1/2 of list between list[mid] and list[last].
   3. If target > list[mid], discard 1/2 of list between list[first] and list[mid].
   4. Continue searching the shortened list until either the target is found, or there are no elements to probe.

Example

```
+---+---+---+---+---+---+---+---+---+---+---+---+
| A | B | C | D | E | F | G | H | I | J | K | L |
+---+---+---+---+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7   8   9   10  11
```

Search for F

low is 0, high is 11.
mid is (high + low) / 2 or (11 + 0) / 2 or 5.
list[5] is F.   // FOUND!!

Search for C

mid is (high + low) / 2 or 5.

list[5] != C.

Since C < list[5], high is mid-1 or 4.

Now search list[0] ... list[4].

mid is (4 + 0) / 2 or 2.

list[2] is C.   // FOUND!!

Since we discard 1/2 of the list with each probe, doubling the size of the list adds only one probe to search time.

But...

- Binary search is fast, but the list must be sorted (ordered).
- Since the list must be ordered, doubling the list substantially increases the total execution time.
- With sorting/searching algorithms, often there is a trade-off in time and space.
- How many probes are needed on an ordered list of size n?

  worst case____________________

  best case_____________________

  average case__________________

Summary
- Searching Techniques
  - Linear (sequential)
    - easy to implement
    - slow
      - best time: one probe
      - average time: n/2 probes
      - worst time: n probes
  - Binary
    - more complicated
    - list must be ordered
    - fast
      - best time: one probe
      - average/worst time: log2n
  - Hash Coding
    - far more complicated
    - works best when list is in random order
    - very fast
      - best time: O(1) or constant time
      - average/worst time: O(1) or constant time
- Sorting Algorithms
  - Insertion Sort
  - Selection Sort
  - Bucket Sort
  - Quicksort
  - Mergesort

Sorting time may depend on:

- initial order of data
- data structure used to represent data
- memory available
- size of record
- sort algorithm
- location of list: internal storage, external device, etc.
- and more

Simple Sorts rearrange data in a list or rearrange pointers in a linked structure.

8. Selection Sort
9. Insertion Sort
10. Bubble Sort (really slow)

Better Sorts, but more involved

11. Shell Sort - improved insertion sort
12. Quicksort - a partition sort (excellent, but limited)
13. Bucket Sort

Selection Sort O(n^2)

Algorithm

0. Find smallest item in list.
1. Exchange it with "first" item in list.
2. Repeat, beginning with the rest of the list.

```
original:
+---+---+---+---+---+---+---+---+
| P | M | G | D | B | E | R | K |
+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7

after first pass:
+---+---+---+---+---+---+---+---+
| B | M | G | D | P | E | R | K |
+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7
```

after second pass:

```
+---+---+---+---+---+---+---+---+
| B | D | G | M | P | E | R | K |
+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7
```

Quicksort   O(nlog2(n))

Quicksort was invented and named by C. A. R. Hoare and is one of the best general-purpose sorting algorithms. It is built on the ideal of partitions, and it uses *divide-and-conquer* strategy. The basic algorithm for a one-dimensional array is as follows.

0. *Partition Step:* Select an element to place in its final position in the array. That is, all the elements to the left will be less than selected element, and all the elements to the right will be greater than the chosen element. We will select the first element in the array and put it in its final place in the array. Then we have one element in its proper location and two unsorted subarrays.

1. *Recursive Step:* Repeat the process on each unsorted subarray.

Each time the partition step is repeated, another element is placed in its final position in the sorted array, and two additional subarrays are created. When a subarray eventually contains only one element, that subarray is sorted and the element is in its final location.

Let's consider the following array of integers.

37   2   6   4   89   8   10   12   68   45

2. Start from the rightmost element in the array and compare each element with the chosen element, 37 here. When an element less than 37 is found (12), swap it with the 37. Now we have
3. 12   2   6   4   89   8   10   37   68   45

4. Start from the left of the array beginning with the element after the 12, and compare each element with 37 until an element greater than 37 is found (89). Then swap 37 with 89. Now we have
5. 12   2   6   4   37   8   10   89   68   45

6. Start from the right but begin with the element before 89. When you find an element less than 37 (10), swap the two elements. Now we have
7. 12   2   6   4   10   8   37   89   68   45

8. Start from the left but begin with the element after 10. When you find an element greater than 37, swap the two elements. Since there are no elements greater than 37, we compare 37 with itself and know that 37 is in its final place in the array. Now we have two unsorted subarrays where the left one is left than 37 and the right one is greater than 37.
9. 12 2 6 4 10 8 37 89 68 45

This is one pass. The sort continues with both subarrays being partitioned in the same manner. The recursive quicksort function is elegant and easy to write. Here is one possibility.

```
quicksort(int array[], int left, int right)
{
  int index;

  if(right > left)  {
    index = partition(array, left, right);
    quicksort(array, left, index-1);
    quicksort(array, index+1, right);
  }
}
```

The partition function is a bit more laborious, but you should be able to write it as a homework lab assignment.

When the original array elements are in random order, quicksort works very well. What happens when the original array elements are sorted? What is the order of the algorithm?